GUITester: Enabling GUI Agents for Exploratory Defect Discovery

Anonymous ACL submission

Abstract

Exploratory GUI testing is essential for software quality but suffers from high manual costs. While Multi-modal Large Language Model (MLLM) agents excel in navigation, they fail to autonomously discover defects due to two core challenges: *Goal-Oriented Masking*, where agents prioritize task completion over reporting anomalies, and *Execution-Bias Attribution*, where system defects are misidentified as agent errors. To address these, we first introduce **GUITestBench**, the first interactive benchmark for this task, featuring 143 tasks across 26 defects. We then propose **GUITester**, a multi-agent framework that decouples navigation from verification via two modules: (i) a *Planning-Execution Module (PEM)* that proactively probes for defects via embedded testing intents, and (ii) a *Hierarchical Reflection Module (HRM)* that resolves attribution ambiguity through interaction history analysis. GUITester achieves an F1-score of 48.90% (Pass@3) on GUITestBench, outperforming state-of-the-art baselines (33.35%). Our work demonstrates the feasibility of autonomous exploratory testing and provides a robust foundation for future GUI quality assurance.

1 Introduction

Exploratory GUI testing is a critical paradigm for ensuring software reliability by uncovering defects within unscripted, complex interaction contexts (Kong et al., 2019; Yu et al., 2025). Unlike script-based testing (Yu et al., 2023), it inherently requires an autonomous navigation of the interface and detection of defects without predefined test oracles (Copche et al., 2023). However, the efficacy of this methodology is traditionally bottlenecked by its heavy reliance on human expertise and subjective judgment, which precludes large-scale, continuous execution in modern rapid-development cycles (Fonseca et al., 2025). While Multimodal Large Language Model (MLLM)-powered GUI agents have demonstrated remarkable proficiency in GUI navigation (Qin et al., 2025; Ye et al., 2025; Zhou et al., 2025), their potential for autonomous defect discovery remains largely unfulfilled.

We identify two fundamental challenges that prevent existing GUI agents from effective exploratory testing: **(i) Goal-Oriented Masking.** Most GUI agents are optimized to maximize task success rates, which inherently encourages robustness against environmental obstacles. In a testing context, this goal-oriented nature leads the agent to perceive functional anomalies as traversable hurdles rather than reportable defects. As shown in Figure 1(a), when encountering a non-responsive button, the agent's policy autonomously seeks alternative navigation paths to reach the goal. This "success-at-all-costs" behavior effectively masks the defect, rendering it invisible to the quality assurance pipeline. **(ii) Execution-Bias Attribution.** Exploratory testing lacks explicit oracles, requiring agents to distinguish between their own operational failures (e.g., coordinate miscalculations) and genuine software defects. Due to the stochastic nature of MLLM interactions, current agents exhibit a systematic bias toward self-attribution: erroneously assuming that any failure to trigger a state change stems from their own execution imprecision. As illustrated in Figure 1(b), GUI-Owl misinterprets a system-level rendering failure as a misaligned click, causing the genuine defect to be misclassified as a transient execution error in the logs.

To investigate the capability of existing GUI agents in exploratory testing scenarios, we introduce **GUITestBench**, the first interactive benchmark for exploratory GUI defect detection. We collect 26 real-world defects across 12 Android applications and construct 143 navigation tasks that encounter these defects during execution. Given defect-agnostic task descriptions, agents must autonomously discover and report defects. To address the aforementioned challenges, we propose **GUI-**

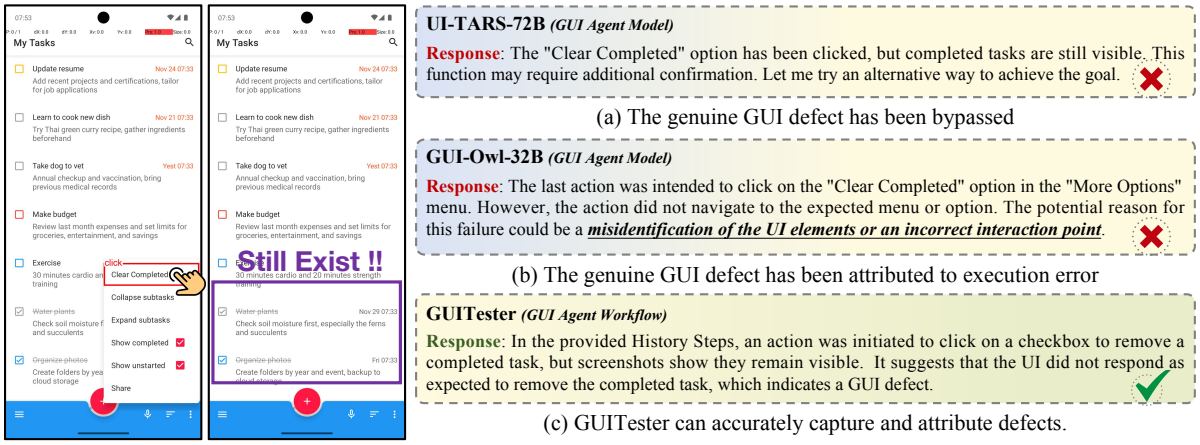


Figure 1: A GUI defect where "Clear Completed" fails to remove the completed tasks. Existing approaches either bypass the defect by attempting alternative paths, or misattribute it to agent execution errors, while GUITester accurately captures the anomaly and attributes it as a genuine GUI defect.

Tester, a multi-agent framework that decouples navigation from defect verification. GUITester employs a **Planning Execution Module (PEM)** that intentionally probes for potential failures, preventing defects from being overlooked by goal-oriented navigation. It further introduces a **Hierarchical Reflection Module (HRM)** that utilizes interaction history to resolve the attribution dilemma, ensuring that software-side defects are not misattributed to agent-side execution slips.

We evaluate GUITester against state-of-the-art GUI automation agents on GUITestBench, including UI-TARS (Qin et al., 2025), GUI-Owl (Ye et al., 2025), and MAI-UI (Zhou et al., 2025). Our results show that existing agents struggle with defect discovery, with the strongest baseline achieving only 33.35% F1-score (Pass@3). In contrast, GUITester significantly improves the F1-score to 48.90%, demonstrating the viability of autonomous exploratory testing and informing future agent design. Our contributions include:

- We define the challenges of task-success bias and attribution ambiguity in MLLM-driven exploratory GUI testing.
- We introduce GUITestBench, an interactive evaluation framework featuring diverse, real-world GUI defects.
- We propose GUITester, a multi-agent framework featuring proactive defect probing and hierarchical reflection, which significantly improves defect discovery rates.

2 Related Works

GUI Agent. Recent advances in multimodal large language models have enabled GUI agents to perform autonomous navigation. Agent workflows such as Mobile-Agent (Wang et al., 2024b,a; Ye et al., 2025) leverage multi-agent collaboration and memory mechanisms for complex multi-step tasks, while AppAgent (Zhang et al., 2023) incorporates document-augmented exploration learning. Meanwhile, Agent models like UI-TARS (Qin et al., 2025; Wang et al., 2025) provide end-to-end GUI interaction capabilities, and GUI-Owl (Ye et al., 2025), MAI-UI (Zhou et al., 2025) enhances navigation through online reinforcement learning. These developments indicate that GUI agents have acquired fundamental capabilities for autonomous navigation, laying the groundwork for downstream applications such as test automation.

Test Automation. Several approaches have applied large language models to GUI testing. AUITestAgent (Hu et al., 2024), GUIPilot (Liu et al., 2025b), and ProphetAgent (Kong et al., 2025) automate specific testing workflows such as test case execution and consistency validation. Temac (Liu et al., 2025a) further introduces multi-agent collaboration for testing tasks. While these approaches show promising results in component-centric testing scenarios, they still rely on predefined test cases. In contrast, exploratory testing, where human testers autonomously navigate applications to discover latent defects without predetermined intents, as in monkey testing, remains largely untouched by GUI agents, being one of the most time-consuming and difficult-to-scale aspects of manual testing.

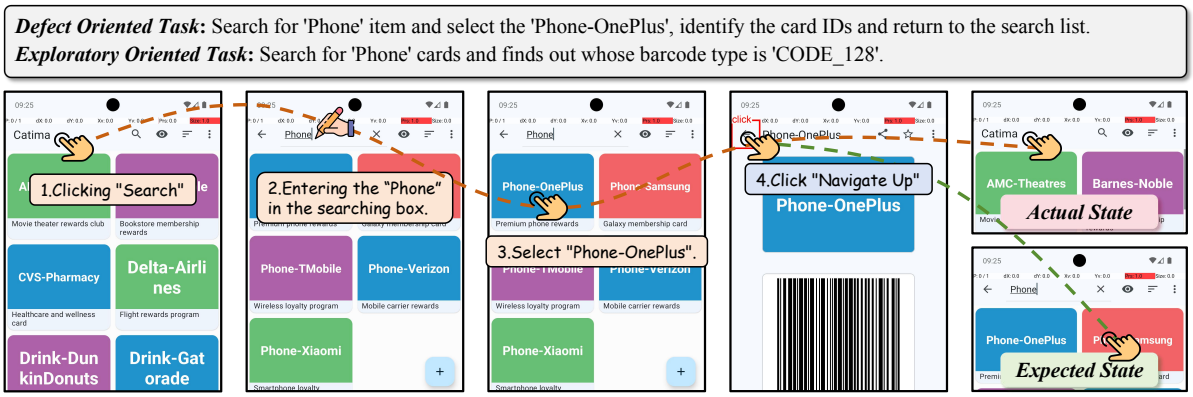


Figure 2: Example of a Navigation Logic Error defect with two task types. The defect-oriented task explicitly guides the agent to trigger the defect, while the exploratory-oriented task may encounter it during exploration. After clicking "Navigate Up", the app returns to the home page (actual) instead of the searching list (expected).

Benchmarks for GUI Testing. Existing benchmarks, such as GTArena (Zhao et al., 2024), focus on evaluating the capabilities of general models in static testing scenarios; however, fail to capture the dynamic nature of real-world applications. There remains a lack of benchmarks specifically designed to evaluate GUI agents' ability to autonomously discover defects through exploration. To fill this gap, we introduce GUITestBench, the first benchmark for evaluating exploratory GUI testing.

3 GUITestBench

GUITestBench is a benchmark that enables agents to interact with mobile apps and discover defects through multi-step operations. This benchmark evaluates three core capabilities for exploratory GUI testing: navigating to defect locations, recognizing anomalous behaviors, and reporting identified defects. We detail the construction process in §3.1 and the evaluation methodology in §3.2.

3.1 Benchmark Construction

We construct GUITestBench through a two-stage process: collecting and categorizing real-world defects (§3.1.1), and synthesizing exploratory tasks with controlled guidance levels (§3.1.2). Dataset statistics are summarized in §3.1.3.

3.1.1 Defect Collection and Categorization

We collect GUI defects from public issues on GitHub, and these defects are categorized into UI functional defects and user experience (UX) defects. Specifically, UI defects stem from implementation errors in specific components, manifesting as failures in the expected functionality of elements (e.g., click failures or incorrect navigation), while UX defects (Baltes and Dashuber, 2024) originate from design flaws in the interaction logic or task flow of multiple components, manifesting as anomalies in the interaction process, which usually cannot be attributed to a single faulty component.

We adopt three fault modes from the defect categories defined in GTArena (Zhao et al., 2024): (1) **Operation No Response (ONR)**: an interaction yields no observable feedback, which applies only to UI defects, as unresponsiveness can be directly attributed to a specific element; (2) **Unexpected Task Result (UTR)**: the outcome deviates from expectations; (3) **Navigation Logic Error (NLE)**: flawed logic causes incorrect navigation flow. More examples are provided in Appendix A.3.

3.1.2 Exploratory Task Synthesis

In practical exploratory GUI testing, defect locations are unknown in advance, making it difficult to evaluate whether the agent's reports are correct. To address this, we control the level of guidance toward defects and propose two synthesis strategies.

Defect-Oriented Task. We manually collect reproduction trajectories that directly guide the agent to trigger defects, and synthesize action-level tasks from these trajectories. By minimizing exploration uncertainty, this strategy isolates the evaluation of defect recognition and reporting capabilities.

Exploration-Oriented Task. We synthesize intent-level tasks where completing the task inevitably passes through the defect location. As shown in Figure 2, the synthesized task requires the agent to explore multiple cards. If the agent navigates correctly, it will encounter the target defect during exploration. By preserving exploration uncertainty, this strategy aims to evaluate the agent's end-to-end

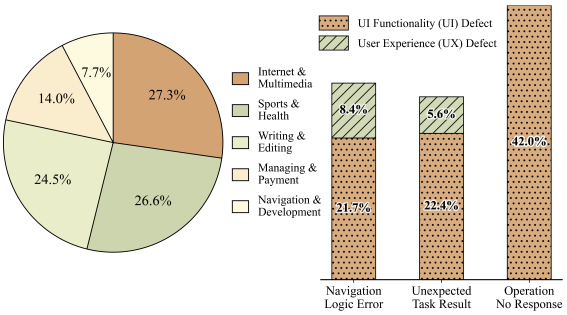


Figure 3: Defects distribution of GUITestBench

defect discovery capability, including navigation, recognition, and reporting.

3.1.3 Dataset Statistics

We collect 26 defect from 12 applications across 5 diverse domains. Using the exploratory task synthesis strategies described above, we expand these scenarios into 143 navigation tasks. The detailed distribution across defect types and application domains is shown in Figure 3. Based on defect-triggering mechanisms, defects fall into two categories: single-action defects, which are triggered by one action on a specific state (62.24%), and multi-action defects, which require a sequence of prerequisite actions (37.76%).

3.2 Benchmark Evaluation

3.2.1 Defect Detection Verification

To evaluate whether the agent successfully triggers a defect, we verify the exploration trajectory against the target defect specifications. Single-action defects have deterministic triggering conditions that can be precisely matched, while multi-action defects involve complex interaction sequences requiring flexible assessment. We thus employ two evaluation approaches:

Rule-based Evaluation. For single-action defects, we verify two conditions: (1) State matching: whether the agent successfully navigates to the screen where the defect resides; (2) Action matching: whether the agent executes the exact defect-triggering action.

Judge-Model Evaluation. For multi-action defects, we employ an LLM as the judge and provide it with detailed defect specifications, as shown in Figure 2, including preconditions, expected results, and screenshots before and after the triggering action. Given the agent's execution trajectory, the judge model determines whether the defect has been successfully triggered. The system prompt is shown in Appendix A.2.

3.2.2 Evaluation Metrics

Based on the above evaluation, we use Recall and F1 score to quantify the overall performance:

Recall measures the proportion of tasks in which the agent successfully identifies the target defect. Since each task corresponds to exactly one issued defect, we define: Recall $= |\mathcal{T}_{detected}|/|\mathcal{T}_{total}|$, where $\mathcal{T}_{detected}$ denotes the set of tasks correctly detecting the defect, and $\mathcal{T}_{total}$ is the set of all tasks.

Precision measures the proportion of tasks that correctly detect defects among all tasks where the agent reports GUI defects: Precision $= |\mathcal{T}_{detected}|/|\mathcal{T}_{declared}|$, where $\mathcal{T}_{declared}$ denotes the set of tasks in which the agent reported GUI defect.

F1 is the harmonic mean of Precision and Recall:

$$\text{F1} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

which provides a balanced measure of the agent's defect discovery capability.

4 GUITester

We propose GUITester, a multi-agent framework that enables GUI agents to exploratory testing. As shown in Figure 4, GUITester comprises two core modules: (1) the Planning Execution Module (PEM, §4.1), which decomposes a navigation task into subtasks with embedded testing intents, guiding the agent to probe potential boundary behaviors; and (2) the Hierarchical Reflection Module (HRM, §4.2), which separates anomaly capture from attribution, ensuring defects are neither bypassed nor misattributed.

4.1 Planning Execution Module

4.1.1 Planner Agent

The Planner decomposes the navigation goal g into a sequence of executable subtasks:

$$\{s_1, s_2, \ldots, s_n\} = \text{Planner}(g, o, h)$$

where o denotes the current observation and h represents the historical context. Each subtask s_i is either a navigation subtask that advances toward the goal, or a test intent that probes potential defects.

Test Intent Generation: Agent navigation tends to follow the shortest path to complete tasks, potentially missing defects hidden in specific interaction contexts. To increase the defect exposure, the Planner embeds test intents that guide the agent to

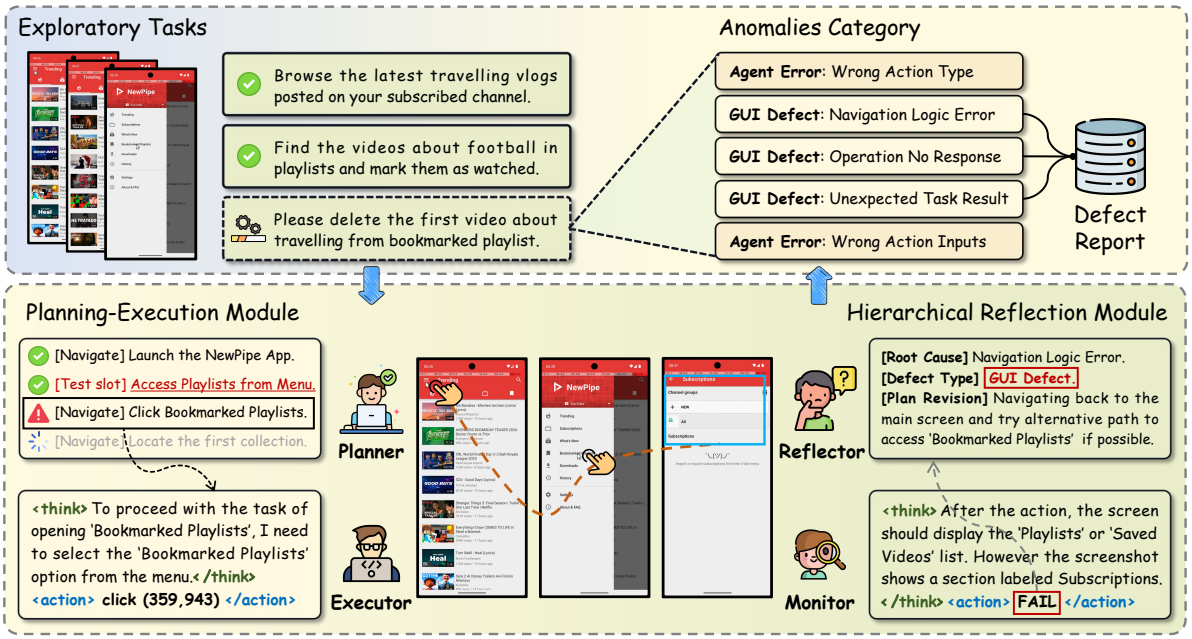


Figure 4: GUITester architecture. The system consists of four agents: (1) a Planner Agent for subtask planning and test intent generation; (2) an Executor Agent for GUI operation; (3) a Monitor Agent for capturing anomalies and controlling the execution process; and (4) a Reflector Agent for anomalies attribution and providing further adjustment for planning suggestions. The workflow are shown in Appendix B.

explore boundary behaviors during navigation. We design three patterns based on how defects manifest in GUI applications, as shown in Appendix 6. These test intents are interleaved with navigation subtasks without disrupting task completion.

4.1.2 Executor Agent

The Executor translates subtasks into executable actions. Given a subtask s_i from the Planner, the Executor observes the current environment state o_t and generates an action:

$$a_t = \text{Executor}(s_i, o_t, \{a_{t'}\}_{t'<t})$$

where $\{a_{t'}\}_{t'<t}$ denotes the action history within the current subtask. We adopt existing GUI agent models (e.g., UI-TARS (Qin et al., 2025), GUI-Owl (Ye et al., 2025)) as the Executor, which generates actions through chain-of-thought reasoning.

4.2 Hierarchical Reflection Module

4.2.1 Monitor Agent

As mentioned above, GUI agents may bypass execution anomalies by exploring alternative paths or waiting for user feedback, potentially overlooking genuine GUI defects. To address this, the Monitor observes the environment's response after each action and determines the execution state:

$$c_t = \text{Monitor}(s_i, o_t, a_t, o_{t+1})$$

where $c_t \in \{\texttt{DONE}, \texttt{FAIL}, \texttt{CONTINUE}\}$. Specifically, the Monitor focuses solely on capturing whether an anomaly occurs without attributing its cause. When anomalies such as unresponsive operations or unexpected state transitions are captured, it terminates the subtask and issues a `FAIL` state, preventing the agent from bypassing potential defects. In contrast, the `CONTINUE` state allows the Executor to proceed with the current subtask, while `DONE` signals the Planner to advance to the next subtask.

4.2.2 Reflector Agent

The Reflector is responsible for attributing anomalies captured by the Monitor. When a `FAIL` state is received, the Reflector analyzes the execution trajectory to distinguish between agent navigation errors and genuine GUI defects:

$$r = \text{Reflector}(s_i, \tau, o_t)$$

where $\tau = \{(o_{t'}, a_{t'})\}_{t'=1}^{t}$ represents the execution trajectory, and r denotes the attribution result.

Visual Attribution. To enable accurate attribution, we visualize the Executor's actions on corresponding screenshots by marking interaction points (Wu et al., 2025; Lu et al., 2024). This allows the Reflector to clearly identify whether the anomaly stems from ineffective operations (e.g., misaligned click) or genuine defects (e.g., unresponsive buttons).

Model	UI-ONR		UI-UTR		UI-NLE		UX-UTR		UX-NLE		Overall	
	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑
Pass@1												
GUI-Owl-7B	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GUI-Owl-32B	1.70	3.10	3.10	5.60	6.50	11.80	0.00	0.00	0.00	0.00	2.80	5.10
MAI-UI-8B	1.13	2.13	6.23	11.60	0.00	0.00	0.00	0.00	0.00	0.00	1.87	3.60
UI-TARS-7B	1.70	3.30	0.00	0.00	3.20	5.30	0.00	0.00	0.00	0.00	1.40	2.60
UI-TARS-72B	16.10	23.65	9.45	15.00	3.23	5.27	0.00	0.00	2.08	2.78	9.68	15.03
UI-TARS-1.5-7B	22.25	28.13	16.68	23.20	9.70	15.33	9.38	12.95	12.53	18.38	16.73	22.95
Mobile-Agent-V3	0.00	0.00	1.03	2.03	3.20	6.00	0.00	0.00	0.00	0.00	0.93	1.83
GUITester *(GUI-Owl-32B)*	16.25	18.45	15.65	17.30	6.45	7.40	0.00	0.00	0.00	0.00	11.80	13.55
GUITester *(UI-TARS-72B)*	21.34	23.94	31.86	35.88	47.74	49.96	0.00	0.00	8.30	9.50	26.60	29.50
GUITester *(UI-TARS-1.5-7B)*	26.68	30.18	34.38	39.10	39.35	44.93	12.50	12.50	12.50	16.70	28.10	32.18
Pass@3												
UI-TARS-72B	28.30	38.20	16.10	21.95	9.70	13.80	6.25	6.65	0.00	0.00	17.95	19.55
UI-TARS-1.5-7B	35.60	42.00	29.00	34.95	16.10	22.45	12.50	**14.30**	16.70	22.20	26.95	33.35
GUITester *(GUI-Owl-32B)*	28.60	29.30	28.10	29.00	12.90	13.30	0.00	0.00	0.00	0.00	21.40	22.10
GUITester *(UI-TARS-72B)*	40.00	40.35	43.80	43.80	**75.80**	**75.80**	0.00	0.00	8.30	9.50	43.40	43.70
GUITester *(UI-TARS-1.5-7B)*	**45.00**	**46.60**	**50.00**	**50.00**	70.00	71.20	**12.50**	12.50	**25.00**	**26.10**	**47.90**	**48.90**

Table 1: GUI defect detection results on GUITestBench. **Bold** and underlined numbers indicate the best scores under the Pass@3 and Pass@1 settings, respectively. Since the improvements of the GUI-Owl, MAI-UI-8B and UI-TARS-7B on Pass@3 are not significant, we have not reported the corresponding results.

Reflection Feedback. After attribution, the Reflector provides feedback to the Planner to prevent repeated failures at the same location. For navigation subtasks attributed to agent errors, the Reflector generates corrective suggestions to guide subsequent planning. For failed test intent subtasks, we prevent them from affecting planning regardless of the attribution result, allowing the agent to continue exploration from the current state.

5 Experiments

We conduct comprehensive experiments to answer two research questions:

- **RQ1**: How do existing GUI agents perform in exploratory GUI testing scenario?
- **RQ2**: How effective is GUITester in addressing the challenges of exploratory GUI testing?

To answer these questions, we design quantitative evaluations on GUITestBench (§5.1, §5.2) and further validate GUITester's practical effectiveness through case studies on released apps (§5.3).

5.1 Experiment Setup

Baseline GUI Agents. We select GUI agents with visual grounding capabilities as our evaluation baselines. These agents can interpret interface states from screenshots and generate corresponding interaction actions. To enable defect reporting during navigation, we augment the action instructions with explicit testing intent through prompt wrapping (see Appendix C for details). We evaluate six open-source GUI agent models from three families: MAI-UI (8B) (Zhou et al., 2025), GUI-Owl (7B/32B) (Ye et al., 2025), and UI-TARS (7B/72B/1.5-7B) (Qin et al., 2025). Additionally, we evaluate Mobile-Agent-V3 (Ye et al., 2025), a multi-agent workflow designed for navigation tasks, powered by GUI-Owl-32B.

GUITester Setup. GUITester comprises four collaborative agents. The **Planner** handles task decomposition and test intent generation, powered by `Qwen3-VL-Plus`. The **Executor** performs GUI actions with a low sampling temperature (0.1) to ensure behavioral stability; we evaluate UI-TARS-72B, UI-TARS-1.5-7B, and GUI-Owl-32B as Executor backbones, respectively. The action space is determined by the Executor model (see Appendix D for detailed action space). The **Monitor** detects anomalies using `GPT-4o`. The **Reflector** performs defect attribution through trajectory analysis, also powered by `Qwen3-VL-Plus`. All experiments are conducted on Android emulators with 1080×2400 resolution.

Evaluation Setup. All models are evaluated on the 143 tasks of GUITestBench with three independent runs. Pass@1 results are computed by averaging the scores across the runs, while Pass@3 indicates

Model	Defect-Ori		Explore-Ori		Single-Act		Multi-Act	
	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑	Recall↑	F1↑
GUI-Owl-7B	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GUI-Owl-32B	5.60	10.30	1.90	3.40	4.30	7.70	0.00	0.00
MAI-UI-8B	3.60	6.37	1.53	3.00	0.00	0.00	3.77	7.00
UI-TARS-7B	2.80	5.10	1.00	1.80	1.10	2.10	1.90	3.60
UI-TARS-72B	16.70	26.70	10.30	16.40	13.50	21.10	9.30	15.40
UI-TARS-1.5-7B	22.90	23.90	14.30	20.40	20.20	28.30	15.40	20.50
Mobile-Agent-V3	0.00	0.00	1.23	2.40	0.00	0.00	1.43	2.77
GUITester *(GUI-Owl-32B)*	19.40	21.50	9.40	10.90	16.50	18.90	6.20	7.00
GUITester *(UI-TARS-72B)*	**33.30**	**35.80**	29.90	33.30	**48.30**	**51.50**	17.90	21.20
GUITester *(UI-TARS-1.5-7B)*	30.60	34.40	**28.00**	**31.20**	43.80	47.00	**18.75**	**23.40**

Table 2: Results across task types and defect complexity.

whether at least one successful detection occurs. We employ `Claude-4-Sonnet` as the Judge.

5.2 Experimental Results

5.2.1 RQ1

Overall Performance. As shown in Table 1, all baseline agents achieve F1 below 25% under the Pass@1 setting, with the best-performing model, UI-TARS-1.5-7B, reaching only 22.95%. Under the Pass@3 setting, this improves to 33.35%, yet over 70% of defects remain undetected. Since Pass@3 reflects the upper bound of detection capability across multiple attempts, these results indicate that exploratory GUI defect discovery remains a challenging task for existing GUI agents. Appendix E provides detaild analysis of failure cases.

UI Defects vs. UX Defects. All baseline agents demonstrate a certain degree of detection capability for UI defects, whereas their performance on UX defects is near zero, with only UI-TARS-1.5-7B achieving non-zero results (F1 of 12.95% on UX-UTR and 18.38% on UX-NLE). This disparity arises from the inherent differences between the two defect categories: UI defects typically manifest as immediate visual anomalies that can be identified through single-frame analysis, while UX defects cannot be attributed to any specific operation and require the agent to comprehend the entire interaction sequence to identify defects. Although existing agents incorporate historical screenshots during inference, their training objectives focus on action prediction, leveraging past information solely to determine the next action, rather than to retrospectively assess the interaction logic.

Analysis Across Defect Types. Across all three defect types (ONR, UTR, and NLE), the baseline agents exhibit consistently poor performance. Despite their distinct manifestations, these defect types share a common requirement: the ability to perceive discrepancies between expected and actual states and to correctly attribute their causes. However, current agents fail to meet this requirement due to task-success bias inherited from their training objectives, which prioritize generating correct actions over verifying environment states. This leads to two failure modes: (1) agents passively accept defective states as normal and continue toward task completion; or (2) when agents do perceive anomalies, they preferentially attribute failures to their own operational errors rather than questioning the GUI itself, causing genuine GUI defects to remain undiscovered.

ANSWER TO RQ1

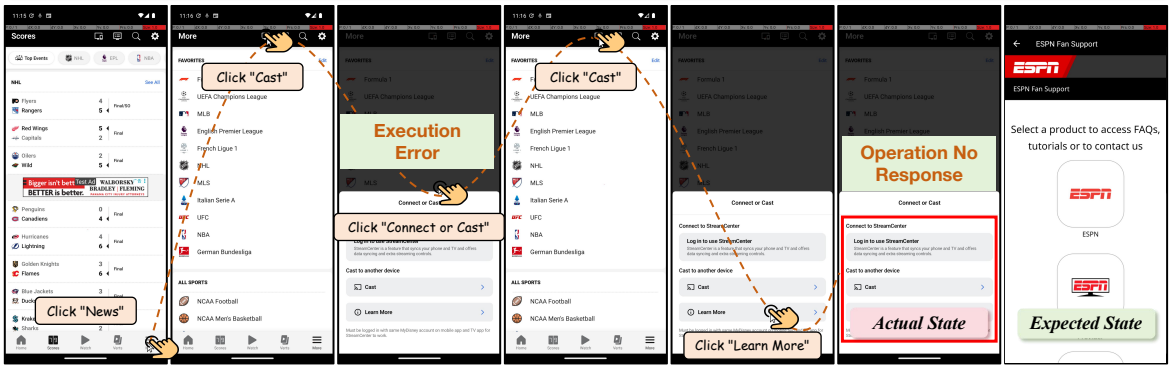
Existing GUI agents struggle with exploratory defect discovery (Recall below 20% at Pass@1). We identify two limitations: (1) the lack of ability to retrospectively analyze interaction logic, and (2) the lack of state verification due to task-success bias. These limitations cause agents to either passively accept defective states or misattribute anomalies to their own execution errors.

5.2.2 RQ2

Overall Performance. As shown in Table 1, under the Pass@3 setting, GUITester achieves significant improvements. For example, with UI-TARS-72B, the overall F1 increases from 19.55% to 43.70%, demonstrating that GUITester can effectively enhance the defect detection capability of GUI agents in exploratory GUI testing.

Analysis Across Task Types. As shown in Table 2, although baseline agents exhibit some detection capability in the Defect-Oriented setting, their performance drops substantially in the Exploration-Oriented setting, indicating that their defect discovery capability is difficult to leverage during

Navigation Task: Navigate to the Where to Watch screen and filter by Leagues to find German 2. Bundesliga matches available on ESPN+.
Subtask, Navigation: Access to Where to Watch. Click the 'More' tab in the bottom navigation bar and look for 'Where to Watch' or similar option.



GUITester Judgement: The goal is to access a menu or list offering streaming or filter options for leagues. The Expected Result specifies that this should be accessible from the "More" tab in the navigation bar. The steps taken involve clicking the "More" tab and repeatedly interacting with the icons and "Learn More" buttons. The result screenshot shows a section labeled "Connect or Cast" with options like "Log in to use StreamCenter," "Cast," and "Learn More," ***which doesn't match the Expected Result, indicating a potential navigation issue***.

Figure 5: A defect detected by GUITester on ESPN (v8.6.0, November 2025). The "Learn More" button in the Cast panel is visually present but functionally non-responsive, failing to navigate to the expected support page.

autonomous exploration. In contrast, GUITester maintains relatively consistent performance across both settings. We attribute this improvement to PEM's test intent embedding, which actively drives the Executor to probe potential boundary behaviors rather than passively waiting to encounter defects, thereby increasing defect exposure during exploration (see Appendix 15, where PEM's boundary testing on a search field uncovered a defect).

Analysis on Defect Complexity. As shown in Table 2, GUITester achieves substantial improvements on Single-Action defects. Single-Action defects manifest immediately after a single operation, making them detectable through per-step state verification. HRM is well-suited for this scenario: the Monitor captures state anomalies right after each action, and the Reflector attributes them before the agent proceeds, ensuring that transient defects are not overlooked or misattributed. On Multi-Action defects, GUITester also outperforms baselines, though with relatively smaller gains. Triggering such defects requires not only anomaly detection and attribution, but also accurate navigation to specific action sequences, placing higher demands on planning and execution capabilities.

ANSWER TO RQ2

GUITester significantly improves detection performance (overall F1 reaching 48.90%). Our results suggest: (1) proactive exploration through embedded test intents, rather than passively waiting to encounter defects; and (2) decoupling anomaly detection from attribution to enable active state verification and defect identification.

5.3 Case Studies on Released APPs

We deploy GUITester on publicly released historical versions of real-world applications to validate its practical effectiveness. Figure 5 shows an ONR defect discovered on ESPN app during exploratory testing. Within the same trajectory, the agent encounters two anomalies: clicking outside the "Connect or Cast" popup causing it to dismiss unexpectedly, and a non-responsive "Learn More" button. HRM correctly attributes the former to agent error and initiates self-correction, while identifying the latter as a GUI defect. This demonstrates that GUITester can accurately attribute anomalies, which is a critical capability for reducing false positives in real-world testing scenarios. More GUI defect detection results are available in Appendix G.

6 Conclusion

This paper identifies two key challenges preventing GUI agents from effective exploratory testing: Goal-Oriented Masking and Execution-Bias Attribution. We introduce GUITestBench, the first interactive benchmark for evaluating defect discovery capabilities, and propose GUITester, a multi-agent framework that proactively probes boundary behaviors and decouples anomaly detection from attribution. Experiments demonstrate that GUITester enables effective defect exposure and accurate defect reporting, validating the feasibility of autonomous exploratory GUI testing and opening up a new direction for GUI agent-based quality assurance.

7 Limitations

While applying GUI agents to exploratory GUI testing opens promising avenues, we find some failure cases caused by practical challenges that warrant further investigation, summarized as follows. Detailed analysis is provided in Appendix F.

(1) **The Wait-or-Miss Dilemma.** Real-world environments are noisy (e.g., network fluctuations, server lag, and page timeouts). We observed GUI agents occasionally reporting slow-loading pages (6-7 seconds) as defects. The naive fix of "just wait longer" creates its own problems: testing efficiency plummets, and worse, some genuine defects manifest as fleeting millisecond glitches that extended waiting would miss entirely. Distinguishing environmental delays from true anomalies remains an open challenge in real-world deployment.

(2) **Monitor Capability Boundaries.** Accurate defect detection requires the Monitor to predict expected states after each action. However, the Monitor's effectiveness is bounded by its domain knowledge and assumptions about application behavior. For instance, it may misjudge valid navigation paths due to unfamiliarity with domain-specific relationships (e.g., UFC as a subsidiary of MMA). Integrating application-specific knowledge or more capable vision-language models could improve precision.

(3) **Small Action Space, Narrow Testing Scope.** Current GUI agents are equipped with action spaces designed for autonomous navigation. However, real-world testing demands richer interactions. Consider a trading application: users routinely `zoom in/out` on stock price charts to examine minute-by-minute fluctuations, yet no existing GUI agent can perform this gesture. Such capability gaps leave some defects unexplored.

(4) **Broader Defect Coverage.** This work focuses on interactive defects. However, GUIs can fail in other ways: overlapping elements, misaligned text, truncated labels. Expanding defect coverage to include such layout issues would enable more comprehensive quality assurance.

Future work will focus on these directions: (1) developing robust strategies to handle real-world environmental noise, (2) enhancing HRM's domain understanding, (3) enriching action spaces for broader testing scenarios, and (4) expanding defect coverage for comprehensive GUI quality assurance.

References

Sebastian Baltes and Veronika Dashuber. 2024. Ux debt: Developers borrow while users pay. *Preprint*, arXiv:2104.06908.

Rubens Copche, Yohan Duarte Pessanha, Vinicius Durelli, Marcelo Medeiros Eler, and Andre Takeshi Endo. 2023. Can a chatbot support exploratory software testing? preliminary results. *Preprint*, arXiv:2307.05807.

Pedro Luís Fonseca, Bruno Lima, and João Pascoal Faria. 2025. Streamlining acceptance test generation for mobile applications through large language models: An industrial case study. *Preprint*, arXiv:2510.18861.

Yongxiang Hu, Xuan Wang, Yingchuan Wang, Yu Zhang, Shiyu Guo, Chaoyi Chen, Xin Wang, and Yangfan Zhou. 2024. Auitestagent: Automatic requirements oriented gui function testing. *Preprint*, arXiv:2407.09018.

Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F. Bissyandé, and Jacques Klein. 2019. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68(1):45–66.

Qichao Kong, Zhengwei Lv, Yiheng Xiong, Dingchun Wang, Jingling Sun, Ting Su, Letao Li, Xu Yang, and Gang Huo. 2025. *ProphetAgent: Automatically Synthesizing GUI Tests from Test Cases in Natural Language for Mobile Apps*, page 174–179. Association for Computing Machinery, New York, NY, USA.

Chenxu Liu, Zhiyu Gu, Guoquan Wu, Ying Zhang, Jun Wei, and Tao Xie. 2025a. Temac: Multi-agent collaboration for automated web gui testing. *Preprint*, arXiv:2506.00520.

Ruofan Liu, Xiwen Teoh, Yun Lin, Guanjie Chen, Ruofei Ren, Denys Poshyvanyk, and Jin Song Dong. 2025b. Guipilot: A consistency-based mobile gui testing approach for detecting application-specific bugs. *Preprint*, arXiv:2506.07385.

Yadong Lu, Jianwei Yang, Yelong Shen, and Ahmed Awadallah. 2024. Omniparser for pure vision based gui agent. *Preprint*, arXiv:2408.00203.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, Wanjun Zhong, Kuanye Li, Jiale Yang, Yu Miao, Woyu Lin, Longxiang Liu, Xu Jiang, Qianli Ma, Jingyu Li, and 16 others. 2025. Ui-tars: Pioneering automated gui interaction with native agents. *Preprint*, arXiv:2501.12326.

Haoming Wang, Haoyang Zou, Huatong Song, Jiazhan Feng, Junjie Fang, Junting Lu, Longxiang Liu, Qinyu Luo, Shihao Liang, Shijue Huang, Wanjun Zhong, Yining Ye, Yujia Qin, Yuwen Xiong, Yuxin Song, Zhiyong Wu, Aoyan Li, Bo Li, Chen Dun, and 93

others. 2025. Ui-tars-2 technical report: Advancing gui agent with multi-turn reinforcement learning. *Preprint*, arXiv:2509.02544.

Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024a. Mobile-agent-v2: mobile device operation assistant with effective navigation via multi-agent collaboration. NIPS '24, Red Hook, NY, USA. Curran Associates Inc.

Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024b. Mobile-agent: Autonomous multi-modal mobile device agent with visual perception. *Preprint*, arXiv:2401.16158.

Qianhui Wu, Kanzhi Cheng, Rui Yang, Chaoyun Zhang, Jianwei Yang, Huiqiang Jiang, Jian Mu, Baolin Peng, Bo Qiao, Reuben Tan, Si Qin, Lars Liden, Qingwei Lin, Huan Zhang, Tong Zhang, Jianbing Zhang, Dongmei Zhang, and Jianfeng Gao. 2025. Gui-actor: Coordinate-free visual grounding for gui agents. *Preprint*, arXiv:2506.03143.

Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, Jitong Liao, Qi Zheng, Fei Huang, Jingren Zhou, and Ming Yan. 2025. Mobile-agent-v3: Fundamental agents for gui automation. *Preprint*, arXiv:2508.15144.

Shengcheng Yu, Chunrong Fang, Yuchen Ling, Chentian Wu, and Zhenyu Chen. 2023. Llm for test script generation and migration: Challenges, capabilities, and opportunities. *Preprint*, arXiv:2309.13574.

Shengcheng Yu, Chunrong Fang, Ziyuan Tuo, Quanjun Zhang, Chunyang Chen, Zhenyu Chen, and Zhendong Su. 2025. Vision-based mobile app gui testing: A survey. *Preprint*, arXiv:2310.13518.

Chi Zhang, Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2023. Appagent: Multimodal agents as smartphone users. *Preprint*, arXiv:2312.13771.

Kangjia Zhao, Jiahui Song, Leigang Sha, Haozhan Shen, Zhi Chen, Tiancheng Zhao, Xiubo Liang, and Jianwei Yin. 2024. Gui testing arena: A unified benchmark for advancing autonomous gui testing agent. *Preprint*, arXiv:2412.18426.

Hanzhang Zhou, Xu Zhang, Panrong Tong, Jianan Zhang, Liangyu Chen, Quyu Kong, Chenglin Cai, Chen Liu, Yue Wang, Jingren Zhou, and Steven Hoi. 2025. Mai-ui technical report: Real-world centric foundation gui agents. *Preprint*, arXiv:2512.22047.

A GUITestBench

A.1 Exploration Task Synthesis

This section details the synthesis procedures for the Defect-Oriented and Exploration-Oriented tasks.

Defect-Oriented Task. Given a manually collected reproduction trajectory that reaches the defect, we employ an LLM to abstract the action sequence into a natural language instruction. The LLM is provided with the application's functional context and the verified interaction sequence, producing a goal-directed instruction that guides the agent directly toward the defect location.

Exploration-Oriented. To construct tasks where the defect serves as a necessary waypoint, we adopt a three-stage synthesis-and-filter pipeline:

- *Pre-defect Intent Synthesis*. Using the reproduction trajectory from the initial state to the defect, we prompt an LLM to generate multiple navigation intents that would lead to the defect location.
- *Post-defect Intent Synthesis*. Starting from the defect page, we prompt an LLM to generate plausible continuation intents, which may navigate deeper into the app, return to previous screens, or explore sibling functionalities.
- *Combination and Filtering*. We combine pre-defect intents, the defect-triggering actions, and post-defect intents into composite task instructions. For instance, 5 pre-defect intents and 3 post-defect intents yield 15 candidate tasks. Each candidate is then executed by a GUI agent, and only those where the agent successfully reaches the defect location are retained.

This pipeline ensures that the resulting tasks possess a bottleneck structure: completing the task necessitates traversing the defect state, enabling evaluation of end-to-end defect discovery under realistic exploration scenarios.

A.2 Multi-Action Defect Verification

Multi-action defects require specific sequences of operations to trigger, making automated verification more challenging than single-action defects. We employ an LLM-based judge to determine whether the agent's trajectory successfully reproduces the target defect. As shown in Table 3, the judge model receives four inputs: (1) the defect description specifying preconditions, trigger actions, and expected results; (2) reference screenshots demonstrating the defect behavior; (3) the agent's execution trajectory; and (4) screenshots of the agent's final state. The judge then performs a three-step verification: checking whether preconditions are satisfied, whether the trigger action is correctly executed, and whether the final state

matches the expected defect behavior. We highlight two key aspects in the prompt design: (1) Strict action sequence matching: A trajectory that only satisfies preconditions but misses the trigger action is marked as failure, since the trigger action is essential for defect manifestation. (2) Flexible input values: The agent may use different input values from the defect description examples, as long as the action sequence and interaction pattern remain consistent.

A.3 More Examples of GUITestBench

We provide representative examples for each defect category in GUITestBench. Each example illustrates both the Defect-Oriented task (with step-by-step guidance) and the Exploration-Oriented task (with only high-level intent), along with the reproduction trajectory and the contrast between actual and expected states. Figure 7 shows a Navigation Logic Error (UI-NLE) where clicking an element leads to an incorrect destination. Figure 8 demonstrates an Operation No Response (UI-ONR) defect where the interface fails to respond as expected. Figure 9 presents a User Experience defect (UX-UTR) where individual operations succeed but the overall task outcome is incorrect. Figure 10 illustrates an Unexpected Task Result (UI-UTR) where user input is not correctly preserved.

B GUITester

Algorithm 1 presents the complete workflow of GUITester. The system operates as a loop coordinated by four agents: **Planner**, **Executor**, **Monitor**, and **Reflector**.

Planning. The Planner decomposes the navigation task g into a sequence of subtasks $\{s_1, \ldots, s_n\}$, which includes both navigation subtasks and test intents (as show in Table 6). If previous failures occurred, the Planner incorporates reflection feedback r to avoid repeating the same errors (Lines 4–5).

Execution. The Executor generates actions a_t based on the current subtask s, observation o_t, and trajectory τ (Lines 28–31). Each action is sent to the environment for execution.

Monitoring. After each action, the Monitor evaluates the execution status $c_t \in \{\texttt{DONE}, \texttt{FAIL}, \texttt{CONTINUE}\}$ by analyzing the environment feedback (Lines 11–20). If the subtask completes successfully (DONE), the system proceeds to the next subtask (Lines 6–10). If a failure is detected (FAIL), control transfers to the Reflector.

Reflection. The Reflector attributes anomalies to either agent execution errors or GUI defects (Lines 21–27). For agent errors within the retry limit, the Executor attempts self-correction. Otherwise, the system resets and generates a new plan to explore alternative paths.

The loop terminates when all subtasks are completed or the maximum retry limit is reached.

Algorithm 1: GUITester Workflow

Input: navigation goal g, current observation o_t, history h, MaxSteps ; // max trajectory steps
Output: a_t or NOOP ; // action, or terminate state
Init: $\tau \leftarrow \emptyset$; // trajectory for the current subtask
replan←True; ; // adjust the current plan
next_subtask←False; ; // get the next subtask
check_status←False; ; // monitoring the state
reflect←False; ; // reflect the failure
send_action←False; ; // send action to env.

SyncState(o_t) ; // share o_t with all agents
2 **while** ¬send_action **do**
// Planner: decompose task into subtasks
3 **if** replan **then**
4 $\{s_1, \ldots, s_n\} \leftarrow$ Planner(g, o_t, h, r);
5 replan ← False, next_subtask ← True;

6 **if** next_subtask **then**
7 $s \leftarrow$ Planner.GetLatestSubtask();
8 **if** $s = \varnothing$ **then**
9 **return** {NOOP} ; // all tasks completed
10 next_subtask ← False;

// Monitor: capture execution anomalies
11 **if** check_status **then**
12 $c_t \leftarrow$ Monitor(s, o_t, τ) ; // {DONE, FAIL, CONTINUE}
13 **if** c_t == DONE **then**
14 $h \leftarrow h \cup \{(s, \texttt{DONE})\}, \tau \leftarrow \emptyset$;
15 next_subtask ← True, check_status ← False;
16 **else if** c_t == FAIL **then**
17 $h \leftarrow h \cup \{(s, \texttt{FAIL})\}$;
18 reflect ← True, check_status ← False;
19 **else**
20 check_status ← False;

// Reflector: attribute anomalies and recovery
21 **if** reflect **then**
22 $r \leftarrow$ Reflector(s, o_t, τ, h) ; // analyze trajectory τ
23 **if** r == AGENT_ERROR $\wedge |\tau| < MaxSteps$ **then**
24 reflect ← False;
; // self-correct without replanning
25 **else**
26 $\tau \leftarrow \emptyset$; // reset trajectory
27 replan ← True, reflect ← False;

// Executor: generate action for current subtask
28 **if** ¬replan ∧ ¬next_subtask ∧ ¬check_status ∧ ¬reflect **then**
29 $a_{t+1} \leftarrow$ Executor(s, o_t, τ); $\tau \leftarrow \tau \cup \{(a_{t+1}, o_t)\}$;
30 check_status ← True;
31 send_action ← True ; // send action to env.

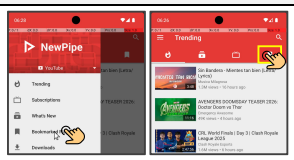
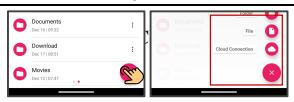
Pattern	Navigation Action	Derived Test Intent
 **Alternative Paths**	Click the hamburger menu icon in the top left corner to open the navigation drawer. In the opened drawer, locate and click the 'Bookmarked Playlists' menu.	***Attempting an alternative navigation path via bookmarked icon.*** Click the bookmarked icon in the top right corner to see if it directly opens Bookmarked Playlists.
Boundary Conditions	Initiate the search for the 'Contact supplier'. Click the magnifying glass search icon in the top right corner, then type the 'Contact supplier' into the search field.	***Search function boundary test.*** Click the search icon, then test entering empty string and special characters like '@@@@@@###$', verify UI response.
 **State Validation**	Tap the 'add' button to initiate folder creation. Click the pink circular '+' button at bottom right corner.	***Check the 'add' button functionality.*** Click the pink circular '+' button at the bottom right corner to confirm it opens the folder creation.

Figure 6: Three test intent patterns designed to increase defect exposure. For each pattern, we show an example navigation action (middle column) and its corresponding test intent (right column) that probes potential boundary behaviors. Alternative Paths explores different entry points to the same functionality; Boundary Conditions tests edge cases such as empty or special character inputs; State Validation verifies UI element responsiveness and state transitions.

C Enabling GUI Agents for Exploratory Testing

Baseline GUI agents are originally designed for navigation tasks without defect detection capabilities. To enable fair evaluation on GUITestBench, we wrap the navigation instructions with explicit testing intent, as shown in Table 4.

The wrapped prompt augments the agent's role from a pure navigator to a navigator with testing awareness. It instructs the agent to: (1) adopt a "test engineer" perspective during navigation, monitoring whether each operation produces expected results; (2) report detected defects in a standardized format without interrupting the navigation task.

To help agents recognize common defect patterns, we provide a checklist covering five categories: incorrect navigation destinations, unresponsive operations, system errors, missing UI elements, and unrelated action results. This checklist is derived from the defect modes in GUITestBench to ensure consistency between the agent's detection scope and the benchmark's evaluation criteria.

D Executor Action Space

UI-TARS (7B/72B/1.5-7B) (Qin et al., 2025) supports `click`, `long_press`, `type`, `scroll`, `open_app`, `drag`, `press_home`, `press_back`, and `finished`; GUI-Owl (7B/32B) (Ye et al., 2025) supports `click`, `long_press`, `swipe`, `type`, `answer`, `system_button`, `wait`, and `terminate`; MAI-UI (8B) (Zhou et al., 2025) supports `click`, `long_press`, `swipe`, `type`, `open`, `drag`, `answer`, `system_button`, `wait`, and `terminate`.

E Detection Failure of GUI Agents

We analyze representative failure cases to understand why existing GUI agents fail to detect defects even when navigating close to defect locations (Figure 11 and 12).

Repetition-Induced Termination. GUI-Owl (Ye et al., 2025) and MAI-UI (Zhou et al., 2025) lack explicit anomaly detection mechanisms. When encountering non-responsive elements, they repeatedly attempt identical actions until the system's termination rule triggers task failure. Without active state verification, these agents cannot distinguish between "action not yet effective" and "action will never be effective due to a defect".

Goal Conflict in Navigation-Oriented Workflows. Mobile-Agent-V3 (Ye et al., 2025) employs planning, execution, and reflection modules all optimized for navigation success. When repurposed for testing, this creates fundamental conflicts: in Task-64, the agent misinterprets the defect as "task not found"; in Task-120, it triggers an unrelated `navigate_home` action. This misalignment causes either premature success or failure declarations without defect reporting.

F Detection Failure of GUITester

While GUITester significantly improves defect detection, we identify two primary failure patterns that suggest directions for future improvement:

Premature Timeout Judgment. The Monitor may misjudge slow-loading states as defects (ONR or UTR) when the environment response time exceeds expectations. As shown in Figure 13, GUITester attempts to open an article in the Zillow app. The page is actually loadable, but fails to fully render within the preset 3-second response buffer. As a result, the screenshot captures the interface in a loading state rather than the final content. After the first timeout, HRM correctly issues a CONTINUE state, allowing the Executor to retry. However, when the second attempt also fails to load within the buffer time, HRM concludes that the link is non-responsive and reports it as a GUI defect. This pattern suggests that incorporating adaptive waiting mechanisms or environment-aware timeout thresholds could reduce such false positives.

Monitor Prediction Errors. Accurate defect detection requires the Monitor to predict expected states after each action. When the Monitor's prediction diverges from the actual expected behavior due to its limited understanding of the application rather than a genuine GUI defect, false positives may occur. As shown in Figure 14, GUITester reports two false defects during a single navigation task in the ESPN app. In the first case, clicking "UFC" navigates to the "MMA" section. Since UFC is a subsidiary of MMA, this navigation is correct, but HRM lacks the domain knowledge to recognize this relationship and misjudges it as a navigation logic error. In the second case, HRM expects the video page to contain accompanying article content, text descriptions, or metadata based on its assumptions about typical video page layouts. When the actual page displays only the video player with an error message, HRM reports this mismatch as a defect. Both cases illustrate that the Monitor's effectiveness is bounded by its domain knowledge and assumptions about application behavior. This limitation suggests that integrating application-specific knowledge or more capable vision-language models could improve precision.

G More Cases on Released APPs

G.1 UI, Unexpected Task result

Figure 15 shows a UTR defect discovered on Fox News app. PEM generates a test intent to probe the search function's boundary behavior by entering special characters "@#$%". The search function should either block such input or return a "no results" message; instead, the app accepts it as valid and returns unrelated content about "Politics". This defect would likely be missed by navigation-oriented exploration, which typically uses meaningful search queries rather than edge-case inputs.

G.2 UX, Unexpected Task result

Figure 16 shows a UX defect discovered on Booking.com app. After editing the email address and navigating back, the keyboard remains visible instead of automatically dismissing. Unlike functional defects with clear error signals, this UX flaw involves improper state transition that does not block task completion but degrades user experience. The Monitor captures this by detecting the mismatch between expected and actual interface states, demonstrating HRM's sensitivity to subtle interaction anomalies.

G.3 UI, Navigation Logic Error

Figure 17 shows a NLE defect discovered on Harbor Freight app. When clicking on "BRAUN 5500 Lumen, 4ft" in the product list, the app navigates to an unrelated product page displaying "BRAUN 1000 Lumen Tactical Rail Mount LED Light". The Monitor detects this navigation logic error by identifying the mismatch between the clicked element and the resulting page content.

G.4 UX, Navigation Logic Error

Figure 18 shows a UX defect discovered on Pinterest app. After opening the "Report Pin" dialog and clicking "Close" to dismiss it, the options menu is unexpectedly closed instead of remaining visible for further actions. Unlike functional defects with explicit error messages, this UX flaw involves unexpected state restoration that does not prevent task completion but disrupts the natural interaction flow. The Monitor captures this by detecting the mismatch between the expected state (options menu visible) and the actual state (menu dismissed).

Defect-Oriented Task: Open the navigation drawer, go to Settings, then select Backup and Restore to manage your song list.
Exploratory-Oriented Task: Go to the settings and back up my current song list.

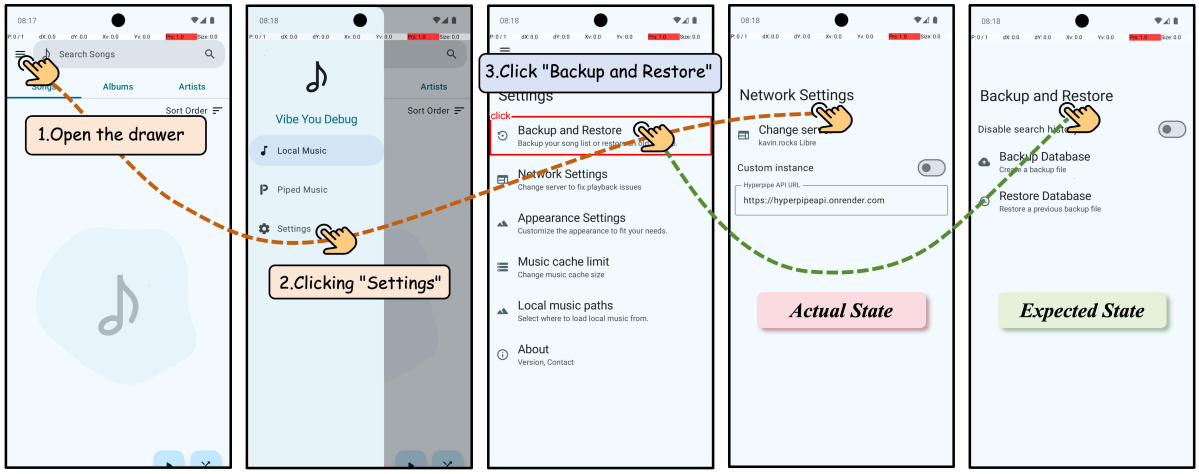


Figure 7: Example of UI-NLE (Navigation Logic Error) defect. The task requires navigating to "Backup and Restore" in Settings. After clicking "Backup and Restore", the app incorrectly navigates to "Network Settings" instead of the expected "Backup and Restore" page, demonstrating a navigation logic error where the destination does not match the triggered element.

Defect-Oriented Task: Open the navigation drawer, go to Settings, select Appearance Settings, then choose Dark to switch to dark mode.
Exploratory-Oriented Task: Switch the app's appearance to dark mode.

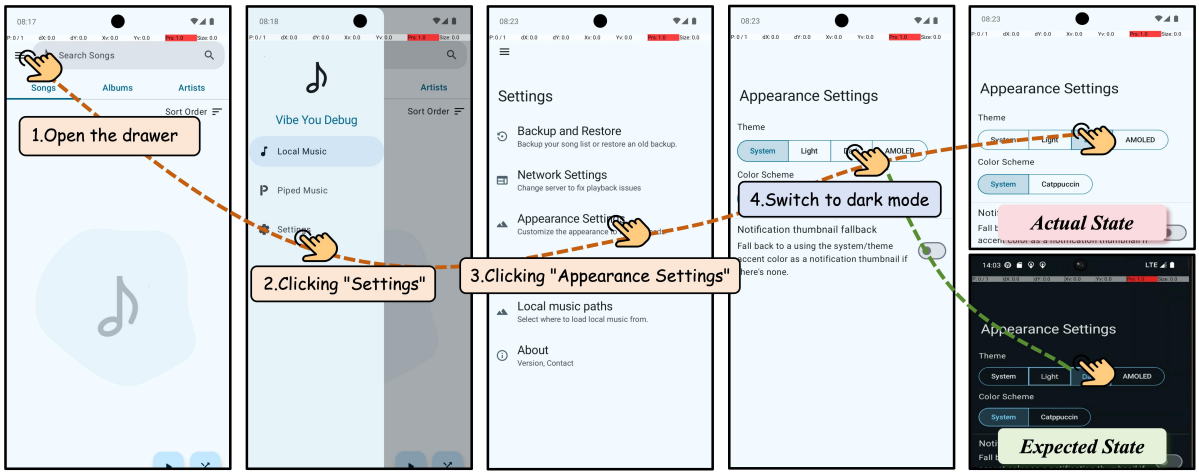


Figure 8: Example of UI-ONR (Operation No Response) defect. The task requires switching the app's appearance to dark mode. After selecting "Dark" in Appearance Settings, the desired appearance does not appear, indicating that the dark mode switch operation did not produce the expected response and the interface state remains inconsistent with user intent.

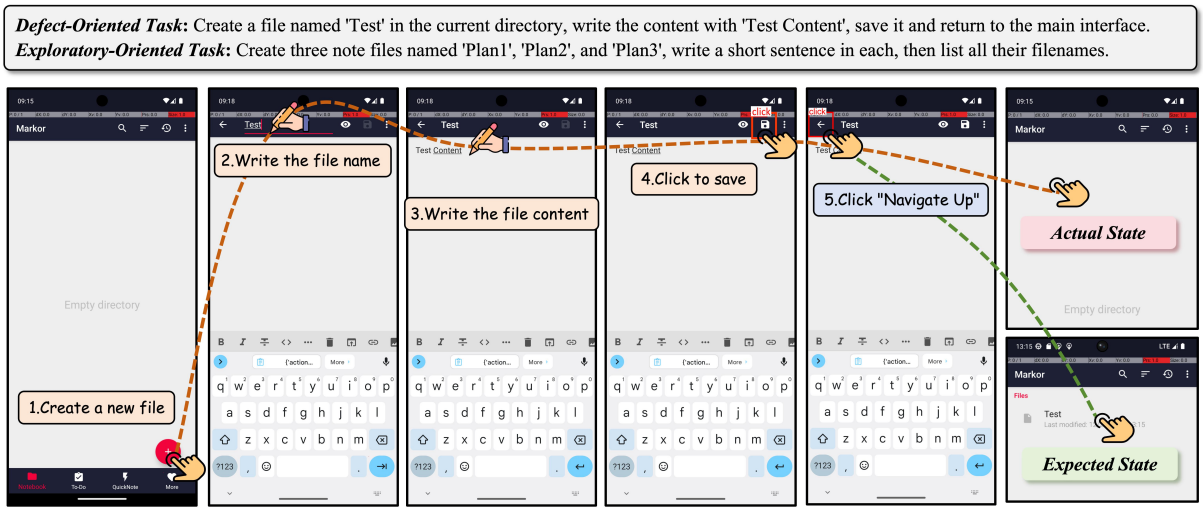


Figure 9: Example of UX-UTR (User Experience, Unexpected Task Result) defect. The task requires creating a file named "Test", writing content, saving it, and returning to the main interface. Although each individual operation succeeds, the final state shows an empty directory without the created file, indicating that the overall task result does not match user expectations despite seemingly correct step-by-step execution.

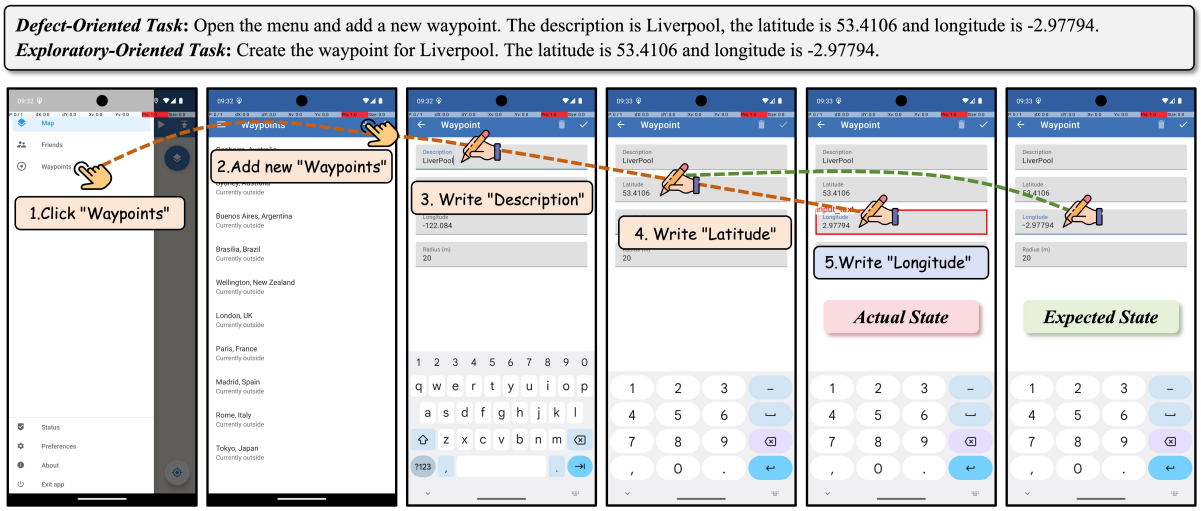


Figure 10: Example of UI-UTR (Unexpected Task Result) defect. The task requires adding a waypoint for Liverpool with specific coordinates (latitude: 53.4106, longitude: -2.97794). After entering all information, the longitude value is incorrectly saved as "2.97794" instead of "-2.97794", demonstrating an unexpected task result where the input data is not correctly preserved.

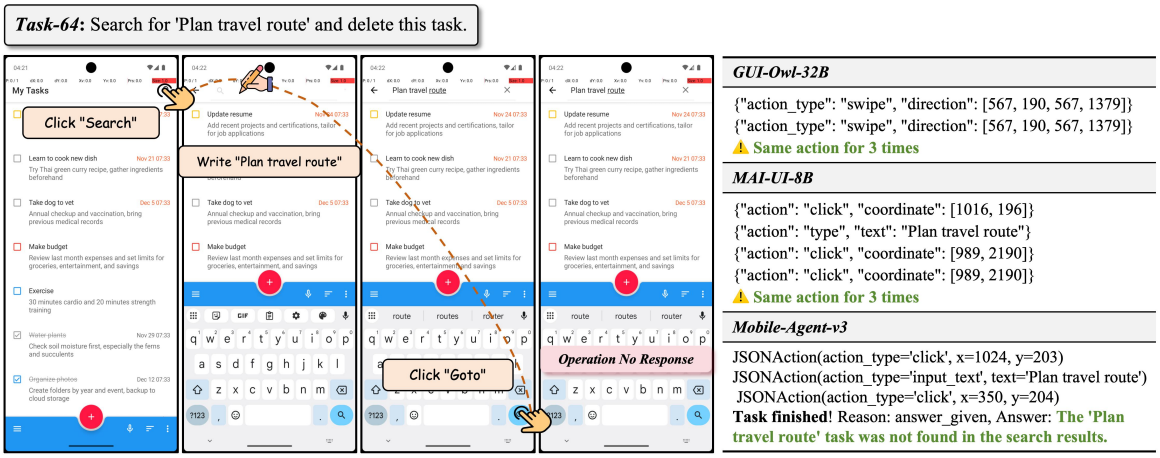


Figure 11: Failure case on Task-64: GUI agents fail to detect an ONR defect in the Tasks app. The task requires searching for "Plan travel route" and deleting it. After navigating to the search results, clicking the "Goto" button produces no response. GUI-Owl-32B repeatedly executes the same swipe action three times without recognizing the anomaly. MAI-UI-8B clicks the target area multiple times but similarly fails to identify the non-responsive state. The preconditions of Mobile-Agent-V3 enabled the model to reach the defect location, but relying solely on the correctness of the trajectory while ignoring environmental feedback caused the model to prematurely declare the task a success.

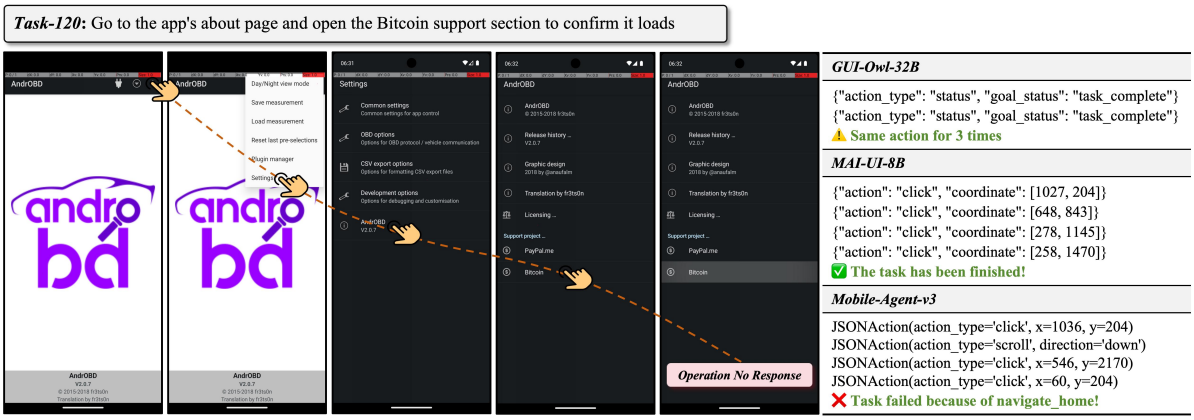
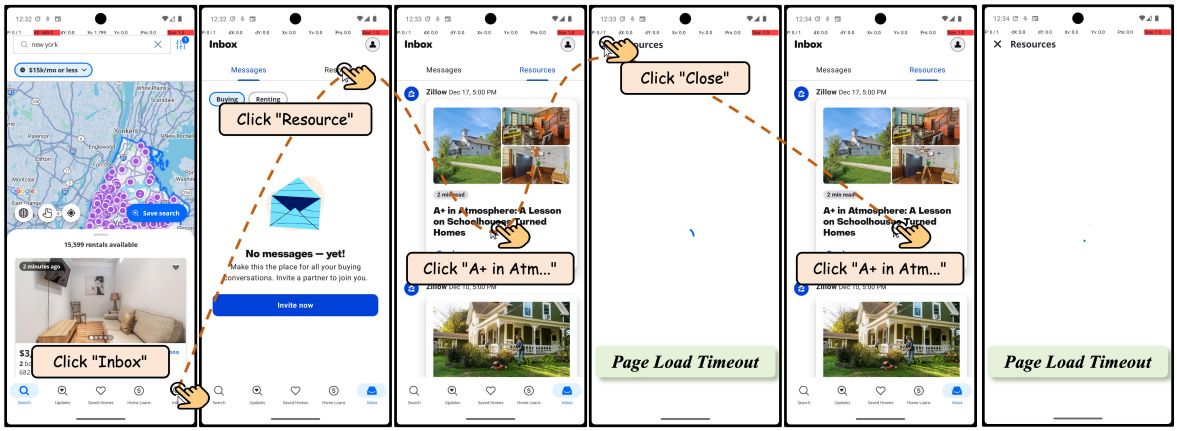


Figure 12: Failure case on Task-120: GUI agents fail to detect an ONR defect in the AndrOBD app. The task requires opening the Bitcoin support section to confirm it loads. When clicking the "Bitcoin" option, the interface fails to respond. GUI-Owl-32B immediately reports "task_complete" after only two attempts, without verifying whether the page actually loaded. MAI-UI-8B successfully navigates to the target location and happens to trigger the defect, but reports task completion rather than identifying the non-responsive state as an anomaly. Mobile-Agent-V3 attempts multiple clicks but ultimately fails due to an unrelated "navigate_home" action, completely missing the defect.

Navigation Task: Go to the Inbox section and switch to Resources tab to read the article about schoolhouses turned into homes.



Current Subtask (step2-5):

Navigation: Open article titled 'schoolhouses turned into homes'. Scroll or search within Resources tab to locate the article and tap it to open.

Executor (UI-TARS-72B)

<think>To proceed with the task of opening the article titled "A+ in Atmosphere: A Lesson on Schoolhouses Turned Homes," I need to interact with the "Read more" link associated with this article. Clicking this link will likely navigate to the full article content, fulfilling the task objective. Tap on the "Read more" link below the article titled "A+ in Atmosphere: A Lesson on Schoolhouses Turned Homes" to open the article.</think>
<action>click(point='<point>500 527</point>')</action>

Executor (UI-TARS-72B)

<think>Since the page content has not yet loaded, it is necessary to wait for the loading process to complete before proceeding. This ensures that I can interact with the relevant elements once they are fully displayed. Wait for the page to finish loading to access the content and proceed with the task.</think>
<action>click(point='<point>66 91</point>')</action>

HRM Judgement: CONTINUE

Expected Result: The task goal is to open the article "schoolhouses turned into homes" with readable content.
Analysis of Last Screenshot: The last screenshot shows the "Read more" link for the article titled "A+ in Atmosphere: A Lesson on Schoolhouses Turned Homes," which matches the task's target article title. The link has been identified visually, but the article detail page itself is not open yet.
Evaluation: (1) The "Read more" link for the correct article is visible, indicating that locating the article has been successful. (2) The article detail page with the readable content is not visible. (3) The required interaction has not yet completed its intended action to navigate to the article page.
Conclusion: The task is **still in progress** as the article detail page has not been opened yet.

Executor (UI-TARS-72B)

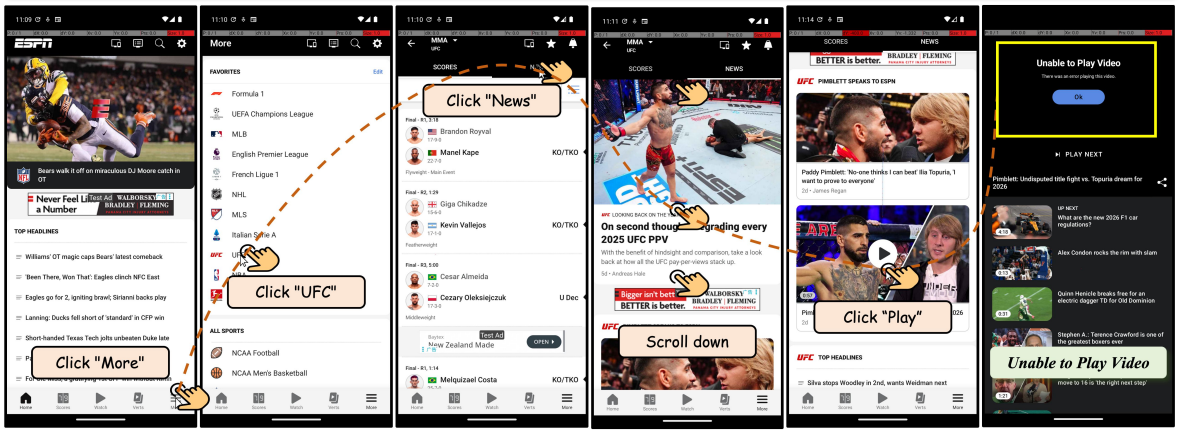
<think>To proceed with the task of opening the article titled "A+ in Atmosphere: A Lesson on Schoolhouses Turned Homes," I need to interact with the "Read more" link associated with this article. Clicking this link will likely navigate to the full article content, fulfilling the task objective.
Tap on the "Read more" link below the article titled "A+ in Atmosphere: A Lesson on Schoolhouses Turned Homes" to open the article.<think>
<action>click(point='<point>567 523</point>')</action>

HRM Judgement: GUI Defect

Expected Result: The article detail page for 'schoolhouses turned into homes' should open with readable content.
Current State: The last screenshot shows a blank page with a "Resources" header, indicating that the expected article content has not loaded.
Analysis: (1) Multiple attempts were made to click the "Read more" link with no visible change in the interface. (2) The interface still shows the "Resources" tab without the article content, suggesting no progress toward the expected result has been achieved.
Decision: The task has failed due to incorrect navigation. Clicking the link failed to load the article, indicating a potential **GUI bug** with loading the content.

Figure 13: A false positive case caused by premature timeout judgment. GUITester attempts to open an article in the Zillow app, but the page fails to load due to network latency. After the first timeout, HRM issues a CONTINUE state and the Executor retries. When the second attempt also times out, HRM reports it as a GUI defect, despite the actual cause being environmental delay rather than a genuine software defect.

Navigation Task: Check the UFC section under Favorites and read news about Paddy Pimblett's undisputed title fight dreams for 2026.



Current Subtask (step0-2):

Navigation: Select UFC section within Favorites. Click the 'UFC' item in the Favorites list to open UFC news and updates.

Executor (UI-TARS-72B)

<think>To proceed with the task of exploring the 'More' tab for UFC or Favorites access, I need to interact with the 'More' tab in the bottom navigation bar. This aligns with the goal of checking the contents of the 'More' tab, as it is the logical next step to uncover additional sections or features.
Tap on the 'More' tab in the bottom navigation bar to explore its contents.</think><action>click(point='<point>899 940</point>')</action>

<think>To proceed with the task of selecting the UFC section within Favorites, I need to click on the "UFC" item in the list. This action will open the UFC news and updates, fulfilling the task objective.Tap on the "UFC" item in the Favorites list to access UFC news and updates.</think>
<action>click(point='<point>200 591</point>')</action>

HRM Judgement: ***GUI Defect***

Expected Result: UFC news feed loads with headlines and articles related to upcoming fights and fighters.
Last Screenshot: Shows MMA UFC section with headlines of fights and details about fighters, indicating the UFC news feed has loaded.
Comparison: The last screenshot shows the MMA news, **which is not matched with expected results.** The news of UFC is invisible, showing only fight results and fighter information.

Current Subtask (step4-5):

Navigation: Scroll through the news list and tap the article titled or mentioning 'Paddy Pimblett's undisputed title fight dreams for 2026'.

Executor (UI-TARS-72B)

<think> To locate the article titled "Paddy Pimblett's undisputed title fight dreams for 2026," I need to scroll down the page to reveal more articles in the list. This action will allow me to continue searching for the desired article. Scroll down the page to display more articles in the list.</think>
<action>scroll(direction='down')</action>

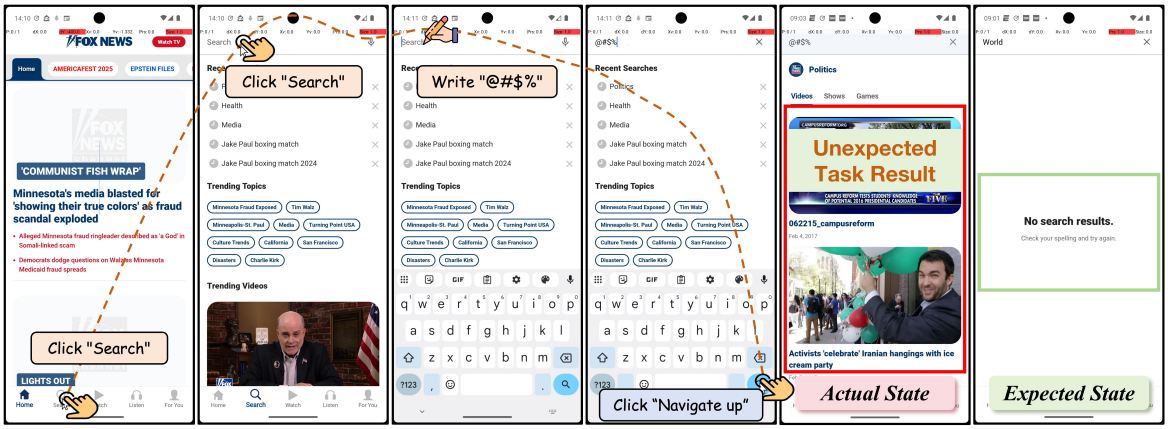
<think>Since the desired article titled "Pimblett: Undisputed title fight vs. Topuria dream for 2026" is now visible on the screen, the next logical step is to tap on it to open and access the content. This action will fulfill the task of locating and opening the specified article. Tap on the article titled "Pimblett: Undisputed title fight vs. Topuria dream for 2026" to open it.</think><action>click(point='<point>515 626</point>')</action>

HRM Judgement: ***GUI Defect***

Expected Result is that the article page opens with full text, images, and metadata about Paddy Pimblett’ s 2026 aspirations.
The last screenshot shows a video error message, "Unable to Play Video", but it does acknowledge the title about Paddy Pimblett's fight.
Analysis: (1) The Expected Result requires accessing the article's content with text, images, and metadata, not just the article's title. (2) **The final state shows an error and no article content visible**; thus, the article has not actually opened or it is not accessible.

Figure 14: False positive cases caused by Monitor prediction errors. GUITester reports two defects in the ESPN app during a single navigation task: (1) clicking "UFC" navigates to "MMA" section, which HRM misjudges as a navigation logic error due to unfamiliarity with the UFC-MMA relationship; (2) the video page lacks article content, which HRM incorrectly expects based on its assumption about typical video page layouts. Both cases stem from the Monitor’s limited domain knowledge rather than genuine GUI defects.

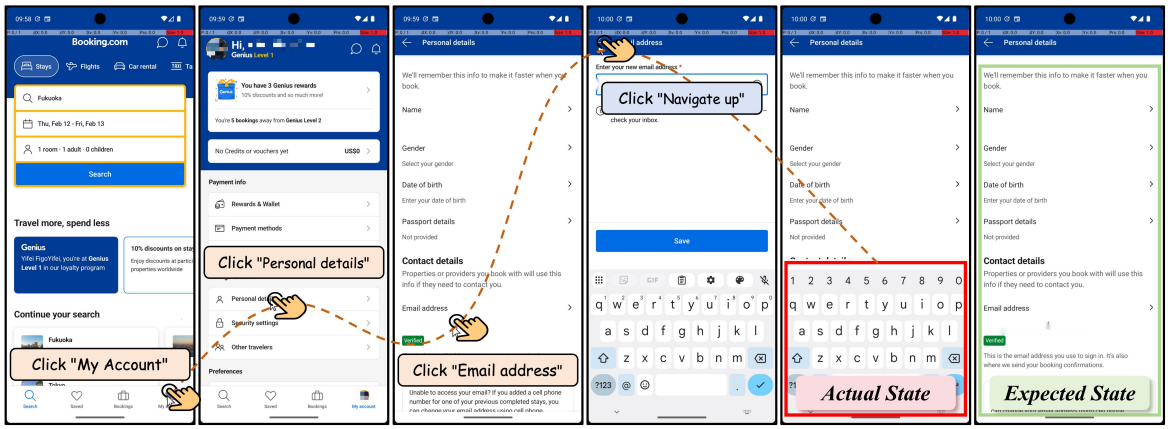
Navigation Task: Search for 'Trump' and browse the latest news articles and videos related to his administration.
Subtask, Test Slot: Search function boundary testing. Enter empty string and special characters (e.g., @#$%) in search field.



Judgement: The Expected Result requires that the search function should handle invalid input (such as special characters) by either blocking it or returning a 'no results' message. In the last screenshot, after entering the special characters "@#$%", the search results page displays content related to different shows. This indicates that the system processed the input and returned results instead of blocking the input ***or explicitly showing a 'no results' message.***

Figure 15: A defect detected by GUITester on Fox News (v5.17.2, December 2025). When entering special characters "@#$%" in the search field, the app returns unrelated content instead of displaying a "no results" message or blocking the invalid input.

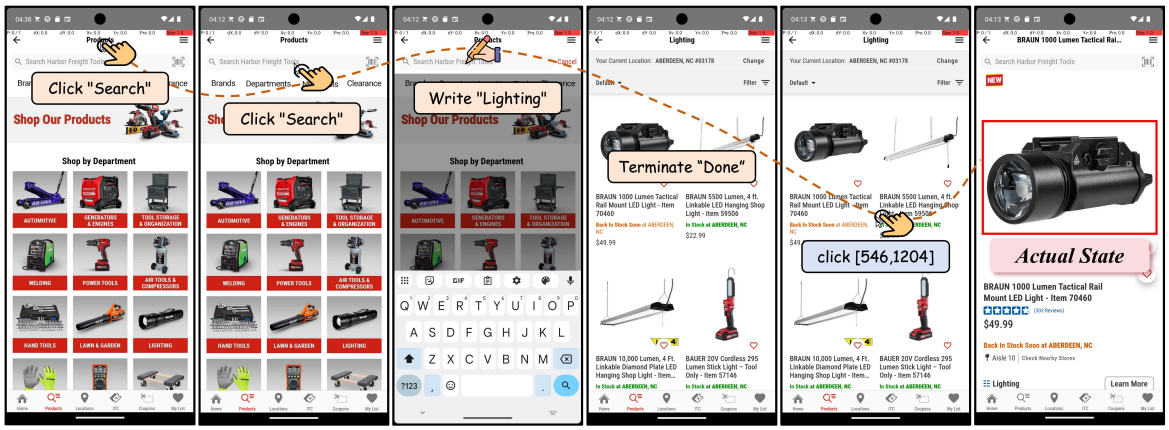
Navigation Task: Check the Privacy and data management settings to review how your personal information is being used.
Subtask, Test Slot: Alternative account access paths. Swipe down or tap menu (if visible) to check for direct 'Privacy Settings' link.



Judgement: Reviewing the steps, the initial intent indicated a need to return to the previous screen to confirm and save the updated email address. The last screenshot shows the "Personal Information" page, which includes input fields for name, gender, date of birth, and a keyboard. The keyboard did not close when the email address was changed. ***This last screenshot demonstrates that the keyboard did not close as expected and remained visible, indicating a UX flaw in the current operation***.

Figure 16: A defect detected by GUITester on Booking.com (v62.1.2, December 2025). After editing the email address in Personal details, the keyboard remains visible when navigating back to the previous screen, instead of automatically dismissing as expected.

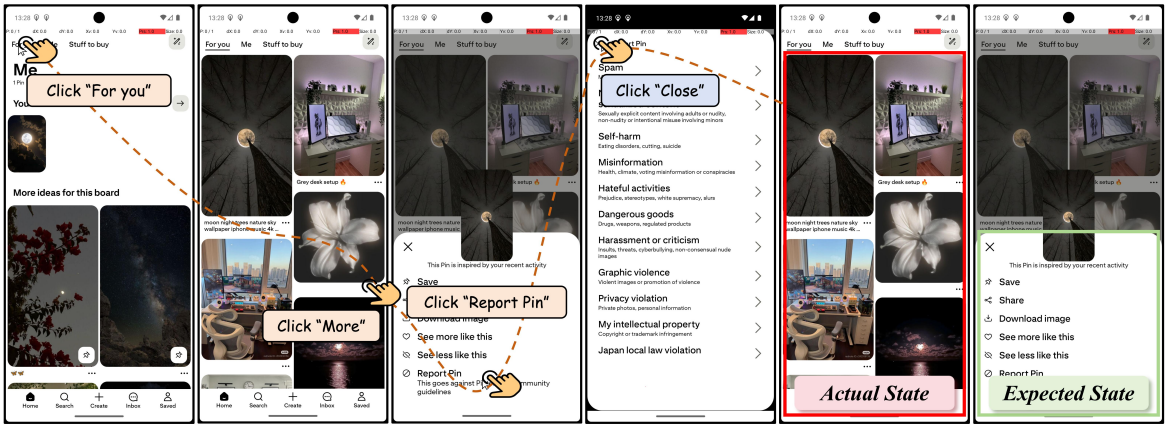
Navigation Task: From the Products page, tap on the Lighting category, then immediately tap the back button.
Subtask, Test Slot: Alternative navigation to Lighting is completed. Use the search bar to type 'Lighting'.



Judgement: Expected result is opening the detailed product page for the target product, BRAUN 5500 Lumen, 4ft. Current screenshot shows the product page, which displays the BRAUN 1000 Lumen Tactical Rail Mount LED light. There is an issue with clicking on the target product. Clicking on the target product, BRAUN 5500 Lumen, 4ft, should display relevant details, but instead leads to an unrelated product page. ***This indicate a potential problem with the button functionality for the target product.***

Figure 17: A defect detected by GUITester on Harbor Freight (v2.5.1, December 2025). Clicking on "BRAUN 5500 Lumen, 4ft" product navigates to an unrelated product page displaying "BRAUN 1000 Lumen Tactical Rail Mount LED Light" instead.

Navigation Task: Browse the "For you" feed and report a pin that violates community guidelines.
Subtask, Test Slot: Test the report flow completion. After selecting a report reason, close the report dialog.



Judgement: The interaction flow navigated to the "For you" feed, click a pin to open the options menu, and selected "Report Pin" to access the report categories. After clicking "Close" to dismiss the report dialog, the options menu should remain visible to allow further actions on the same pin. The last screenshot demonstrates that the options menu was unexpectedly dismissed after closing the report dialog, forcing the user to re-trigger the menu if additional actions are needed, indicating a flaw in the current operation.

Figure 18: A defect detected by GUITester on Pinterest (v13.46.1, December 2025). After opening the "Report Pin" dialog and clicking "Close" to dismiss it, the options menu is unexpectedly closed instead of remaining visible for further actions on the same pin.

You are a GUI Defect Verification Expert. Your task is to determine whether the agent's exploration trajectory has successfully reproduced a known GUI defect.
You'll receive the following information:

- **Defect Description**: A detailed description of the known GUI defect, including its preconditions, trigger action, and expected result.
- **Defect Behavior Reference Images**: Screenshots showing the defect behavior.
- **Agent's Trajectory**: A sequence of actions and final observations performed by the agent.
- **Agent's Trajectory Images**: Screenshots showing the agent's final state.

Your Task
Analyze the agent's trajectory and determine whether the defect was successfully reproduced.
Verification Checklist
You MUST explicitly verify each of the following:

1. **Precondition Check**: Did the agent correctly establish ALL preconditions?
2. **Trigger Action Check**: Did the agent perform the correct trigger action?
3. **Result Check**: Does the final state match the known defect behavior?
4. **Final Verdict**:
   (a) **GUI_BUG**: Agent successfully reproduced the defect.
   (b) **EXECUTOR_ERROR**: The agent failed to reproduce the defect.

Important Note

1. **Action Sequence Matters**: A trajectory that only sets up preconditions but MISSES the trigger action should be marked as **EXECUTOR_ERROR**. The trigger action is the critical step that actually causes the defect to manifest.
2. **Input Values May Vary**: The defect description uses specific examples (e.g., "Pay Bills") for illustration purposes. The agent may use different input values (e.g., "XYZ123"), which is acceptable. However, the **action sequence and interaction pattern** must match the defect description exactly.
3. **Be Strict About Completeness**: Do not assume or infer actions that are not explicitly documented in the trajectory. If a required action is missing from the trajectory, it was not performed.

Output Format

1. Please provide a clear and concise thought within `<think>` and `</think>` tags.
2. Please provide the final verdict within `<answer>` and `</answer>` tags.
3. The final verdict must be either **GUI_BUG** or **EXECUTOR_ERROR**.
4. Here is a sample output: `<think>`your thought`</think><answer>`final verdict`</answer>`.

Table 3: Judge Model Prompt

You are a GUI Agent with defect detection capabilities. While completing the navigation task, you need to explore and examine each interface from the perspective of a "test engineer":

- Before and after each execution: Consider whether the model's response matches expectations.
- Throughout: Pay attention to anything that seems "off".

Bug Reporting Requirements

- When you find a potential defect, output it as {"action": "answer", "text": "GUI_BUG"}.
- Do not interrupt the main task because of finding a defect.

The name of current app is {`app_name`}.
The user query: {`instruction`}.
Task progress (You have done the following operation on the current device): {`histories`}.

The following tips can help you complete user tasks:

1. Wrong Destination / Incorrect Navigation
   - Action intended to open/navigate to X, but system opened/navigated to Y instead.
2. Action Had No Effect After Multiple Attempts
   - The same action was repeated 2+ times with no state change.
   - The UI remains completely unchanged despite valid interaction attempts.
3. Error State or System Failure
   - An error message appeared and blocked progress.
   - The interface crashed, froze, or entered an unrecoverable state.
   - The app closed unexpectedly.
4. Required Element Permanently Missing
   - The target UI element does not exist and cannot be accessed through any reasonable path.
   - The feature appears to be unimplemented or broken.
5. Completely Unrelated Result
   - The action triggered functionality that is logically unrelated to the intent.

Table 4: Exploratory GUI Testing Wrapped Prompt